第 3 章　关联规则挖掘

3.1　基本概念

关联规则挖掘发现大量数据中项集之间有趣的关联联系。如果两项或多项属性之间存在关联，那么其中一项的属性就可以依据其他属性值进行预测。关联规则挖掘是数据挖掘中的一个重要的课题，最近几年已被业界深入研究和广泛应用。

关联规则研究有助于发现交易数据库中不同商品(项)之间的联系，找出顾客购买行为模式，如购买了某一商品对购买其他商品的影响。分析结果可以应用于商品货架布局、货存安排以及根据购买模式对用户进行分类。

关联规则挖掘问题可以分为两个子问题：第一个是找出事务数据库中所有大于等于用户指定的最小支持度的数据项集；第二个是利用频繁项集生成所需要的关联规则，根据用户设定的最小置信度进行取舍，最后得到强关联规则。识别或发现所有频繁项目集是关联规则发现算法的核心，关联规则的基本描述如下：

1. 项与项集

数据库中不可分割的最小单位信息称为项(或项目)，用符号 i 表示，项的集合称为项集。设集合 $I=\{i_1,i_2,\cdots,i_k\}$ 是项集，I 中项目的个数为 k，则集合 I 称为 k-项集。例如，集合{啤酒，尿布，奶粉}是一个 3-项集。

2. 事务

设 $I=\{i_1,i_2,\cdots,i_k\}$ 是由数据库中所有项目构成的集合，事务数据库 $T=\{t_1,t_2,\cdots,t_n\}$ 是由一系列具有唯一标识的事务组成的。每一个事务 $t_i(i=1,2,\cdots,n)$ 包含的项集都是 I 的子集。例如，顾客在商场里同一次购买多种商品，这些购物信息在数据库中有一个唯一的标识，用以表示这些商品是同一顾客同一次购买的，称该用户的本次购物活动对应一个数据库事务。

3. 项集的频数(支持度计数)

包括项集的事务数称为项集的频数(支持度计数)。

4. 关联规则

关联规则是形如 $X\Rightarrow Y$ 的蕴涵式，其中 X,Y 分别是 I 的真子集，并且 $X\cap Y=\phi$。X 称为规则的前提，Y 称为规则的结果。关联规则反映 X 中的项目出现时，Y 中的项目也跟着出现的规律。

5. 关联规则的支持度(support)

关联规则的支持度是交易集中同时包含 X 和 Y 的交易数与所有交易数之比,它反映了 X 和 Y 中所含的项在事务集中同时出现的频率,记为 support($X \Rightarrow Y$),即

$$\text{support}(X \Rightarrow Y) = \text{support}(X \bigcup Y) = P(XY) \tag{3-1}$$

6. 关联规则的置信度(confidence)

关联规则的置信度是交易集中包含 X 和 Y 的交易数与所有交易数与包含 X 的交易数之比,记为 confidence($X \Rightarrow Y$),置信度反映了包含 X 的事务中出现 Y 的条件概率。即

$$\text{confidence}(X \Rightarrow Y) = \frac{\text{support}(X \bigcup Y)}{\text{support}(X)} = P(Y \mid X) \tag{3-2}$$

7. 最小支持度与最小置信度

通常用户为了达到一定的要求,需要指定规则必须满足的支持度和置信度阈值,此两个值称为最小支持度阈值(min_sup)和最小置信度阈值(min_conf)。其中,min_sup 描述了关联规则的最低重要程度,min_conf 规定了关联规则必须满足的最低可靠性。

8. 强关联规则

support($X \Rightarrow Y$)$\geqslant$min_sup 且 confidence($X \Rightarrow Y$)$\geqslant$min_conf,称关联规则 $X \Rightarrow Y$ 为强关联规则,否则称 $X \Rightarrow Y$ 为弱关联规则。通常所说的关联规则一般是指强关联规则。

9. 频繁项集

设 $U \subseteq I$,项目集 U 在数据集 T 上的支持度是包含 U 的事务在 T 中所占的百分比,即

$$\text{support}(U) = \frac{\| \{t \in T \mid U \subseteq t\} \|}{\| T \|} \tag{3-3}$$

式中,$\| \cdot \|$ 表示集合中的元素数目。对项目集 I,在事务数据库 T 中所有满足用户指定的最小支持度的项目集,即不小于 min_sup 的 I 的非空子集,称为频繁项目集或大项目集。

10. 项目集空间理论

Agrawal 等人建立了用于事务数据库挖掘的项目集空间理论。理论的核心为:频繁项目集的子集仍是频繁项目集;非频繁项目集的超集是非频繁项目集。

3.2 关联规则挖掘算法——Apriori 算法原理

最著名的关联规则发现方法是 R. Agrawal 提出的 Apriori 算法。

1. Apriori 算法的基本思想

Apriori 算法的基本思想是通过对数据库的多次扫描来计算项集的支持度,发现所有

的频繁项集从而生成关联规则。Apriori 算法对数据集进行多次扫描。第一次扫描得到频繁 1-项集的集合 L_1，第 $k(k>1)$次扫描首先利用第$(k-1)$次扫描的结果 L_{k-1} 来产生候选 k-项集的集合 C_k，然后在扫描的过程中确定 C_k中元素的支持度，最后在每一次扫描结束时计算频繁 k-项集的集合 L_k，算法在当候选 k-项集的集合 C_k为空时结束。

2. Apriori 算法产生频繁项集的过程

产生频繁项集的过程主要分为连接和剪枝两步：

(1) 连接步。为找到 $L_k(k\geqslant 2)$，通过 L_{k-1}与自身作连接产生候选 k-项集的集合 C_k。设 l_1 和 l_2 是 L_{k-1}中的项集。记 $l_i[j]$ 表示 l_i 的第 j 个项。Apriori 算法假定事务或项集中的项按字典次序排序；对于$(k-1)$项集 l_i，对应的项排序为 $l_i[1]<l_i[2]<\cdots<l_i[k-1]$。如果 L_{k-1} 的元素 l_1 和 l_2 的前$(k-2)$个对应项相等，则 l_1 和 l_2 可连接。即如果$(l_1[1]=l_2[1])\cap(l_1[2]=l_2[2])\cap\cdots\cap(l_1[k-2]=l_2[k-2])\cap(l_1[k-1]<l_2[k-1])$时，$l_1$ 和 l_2 可连接。条件 $l_1[k-1]<l_2[k-1]$ 可以保证不产生重复，而按照 $L_1,L_2,\cdots,L_{k-1},L_k,\cdots,L_n$ 次序寻找频繁项集可以避免对事务数据库中不可能发生的项集所进行的搜索和统计工作。连接 l_1 和 l_2 产生的结果项集为$(l_1[1],l_1[2],\cdots,l_1[k-1],l_2[k-1])$。

(2) 剪枝步。由 Apriori 算法的性质可知，频繁 k-项集的任何子集必须是频繁项集。由连接生成的集合 C_k 需要进行验证，去除不满足支持度的非频繁 k-项集。

3. Apriori 算法的主要步骤

(1) 扫描全部数据，产生候选 1-项集的集合 C_1。

(2) 根据最小支持度，由候选 1-项集的集合 C_1产生频繁 1-项集的集合 L_1。

(3) 对 $k>1$，重复执行步骤(4)、(5)、(6)。

(4) 由 L_k执行连接和剪枝操作，产生候选$(k+1)$-项集的集合 C_{k+1}。

(5) 根据最小支持度，由候选$(k+1)$-项集的集合 C_{k+1}，产生频繁$(k+1)$-项集的集合 L_{k+1}。

(6) 若 $L\neq\phi$，则 $k=k+1$，跳往步骤(4)；否则，跳往步骤(7)。

(7) 根据最小置信度，由频繁项集产生强关联规则，结束。

4. Apriori 算法描述

输入：数据库 D，最小支持度阈值 min_sup。

输出：D 中的频繁集 L。

(1) Begin

(2) L_1=1-频繁项集；

(3) for($k=2;L_{k-1}\neq\phi;k++$)do begin

(4) C_k=Apriori_gen(L_{k-1})；{调用函数 Apriori_gen(L_{k-1})通过频繁$(k-1)$-项集产生候选 k-项集}

(5) for 所有数据集 $t\in D$ do begin {扫描 D 用于计数}

(6) C_t=subset(C_k,t);{用 subset 找出该事务中候选的所有子集}
(7) for 所有候选集 $c \in C_t$ do
(8) c. count++;
(9) end;
(10) $L_k=\{c \in C_k \mid c.\ \text{count} \geqslant \text{min\_sup}\}$
(11) end
(12) end
(13) Return $L_1 \cup L_2 \cup L_k \cup \cdots \cup L_m$ {形成频繁项集的集合}

3.3 Apriori 算法实例分析

【例 3.1】 表 3-1 是一个数据库的事务列表,在数据库中有 9 笔交易,即|D|=9。每笔交易都用唯一的标识符 TID 作标记,交易中的项按字典序存放,下面描述一下 Apriori 算法寻找 D 中频繁项集的过程。

表 3-1 数据库的事务列表

事 务	商品 ID 的列表	事 务	商品 ID 的列表
T100	I_1,I_2,I_5	T600	I_2,I_3
T200	I_2,I_4	T700	I_1,I_3
T300	I_2,I_3	T800	I_1,I_2,I_3,I_5
T400	I_1,I_2,I_4	T900	I_1,I_2,I_3
T500	I_1,I_3		

设最小支持度计数为 2,即 min_sup=2,利用 Apriori 算法产生候选项集及频繁项集的过程如下所示。

1) 第一次扫描

扫描数据库 D 获得每个候选项的计数:

C_1

项集	支持度计数
{I_1}	6
{I_2}	7
{I_3}	6
{I_4}	2
{I_5}	2

比较候选支持计数与最小支持度计数 →

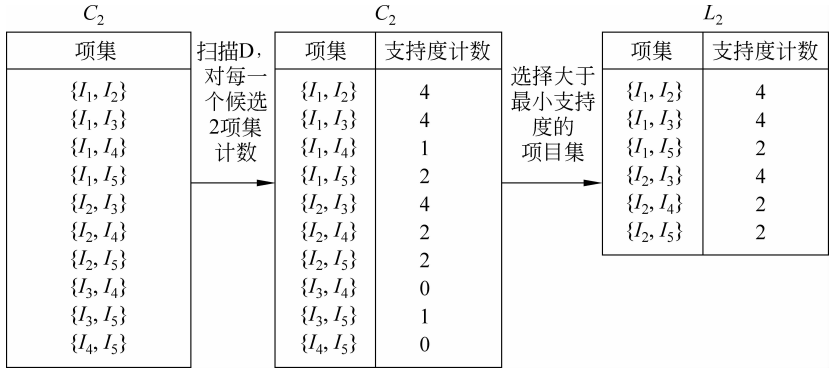
L_1

项集	支持度计数
{I_1}	6
{I_2}	7
{I_3}	6
{I_4}	2
{I_5}	2

由于最小事务支持数为 2,没有删除任何项目。可以确定频繁 1-项集的集合 L_1,它由具有最小支持度的候选 1-项集组成。

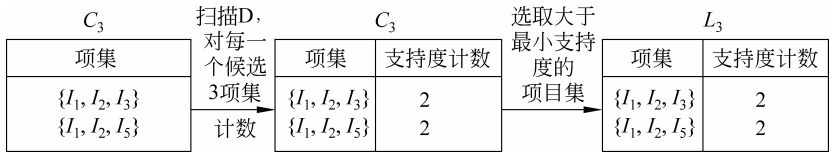
2）第二次扫描

为发现频繁2-项集的集合L_2，算法使用$L_1 \infty L_1$产生候选2-项集的集合C_2。在剪枝步没有候选从C_2中删除，因为这些候选的每个子集也是频繁的。



3）第三次扫描

$L_2 \infty L_2$产生候选3-项集的集合C_3。



候选3项集C_3产生的详细地列表如下：

（1）连接$C_3 = L_2 \infty L_2$

$$= \{\{I_1, I_2\}, \{I_1, I_3\}, \{I_1, I_5\}, \{I_2, I_3\}, \{I_2, I_4\}, \{I_2, I_5\}\} \infty \{\{I_1, I_2\}, \{I_1, I_3\}, \{I_1, I_5\}, \{I_2, I_3\}, \{I_2, I_4\}, \{I_2, I_5\}\}$$

$$= \{\{I_1, I_2, I_3\}, \{I_1, I_2, I_5\}, \{I_1, I_3, I_5\}, \{I_2, I_3, I_4\}, \{I_2, I_3, I_5\}, \{I_2, I_4, I_5\}\}。$$

（2）使用Apriori性质剪枝：频繁项集的所有非空子集也必须是频繁的。例如$\{I_1, I_3, I_5\}$的2-项子集是$\{I_1, I_3\}$，$\{I_1, I_5\}$和$\{I_3, I_5\}$。$\{I_3, I_5\}$不是L_2的元素，因而不是频繁的。因此，从C_3中删除$\{I_1, I_3, I_5\}$。剪枝$C_3 = \{\{I_1, I_2, I_3\}, \{I_1, I_2, I_5\}\}$。

4）第四次扫描

算法使用$L_3 \infty L_3$产生候选4-项集的集合C_4。$L_3 \infty L_3 = \{\{I_1, I_2, I_3, I_5\}\}$，根据Apriori性质，因为它的子集$\{I_2, I_3, I_5\}$不是频繁的，所以这个项集被删除。这样$C_4 = \phi$，因此算法终止，找出了所有的频繁项集。

【例3.2】 表3-2为某一超市销售事务数据库D，使用Apriori算法发现D中的频繁项目集。

事务数据库D中有9个事务，即$\| D \| = 9$。超市中有5件商品可供顾客选择，即$I = \{I_1, I_2, I_3, I_4, I_5\}$，且$\| I \| = 5$。设最小支持数minsup_count=2，则对应的最小支持度为2/9=2.2%。

表 3-2　某一超市销售事务数据库

TID	商品 ID 列表	TID	商品 ID 列表
T100	I_1, I_2, I_5	T600	I_2, I_3
T200	I_2, I_4	T700	I_1, I_3
T300	I_2, I_3	T800	I_1, I_2, I_3, I_5
T400	I_1, I_2, I_4	T900	I_1, I_2, I_3
T500	I_1, I_3		

寻找所有频繁项集的过程如下。

C_1

项集	支持度计数
$\{I_1\}$	6
$\{I_2\}$	7
$\{I_3\}$	6
$\{I_4\}$	2
$\{I_5\}$	2

扫描D，对每一候选1项集计数 →

选取大于最小支持度的项目集候选1项计数 →

L_1

项集	支持度计数
$\{I_1\}$	6
$\{I_2\}$	7
$\{I_3\}$	6
$\{I_4\}$	2
$\{I_5\}$	2

C_2

项集
$\{I_1, I_2\}$
$\{I_1, I_3\}$
$\{I_1, I_4\}$
$\{I_1, I_5\}$
$\{I_2, I_3\}$
$\{I_2, I_4\}$
$\{I_2, I_5\}$
$\{I_3, I_4\}$
$\{I_3, I_5\}$
$\{I_4, I_5\}$

由L_1产生C_2 →

扫描D，对每一候选2项集计数 →

C_2

项集	支持度计数
$\{I_1, I_2\}$	4
$\{I_1, I_3\}$	4
$\{I_1, I_4\}$	1
$\{I_1, I_5\}$	2
$\{I_2, I_3\}$	4
$\{I_2, I_4\}$	2
$\{I_2, I_5\}$	2
$\{I_3, I_4\}$	0
$\{I_3, I_5\}$	1
$\{I_4, I_5\}$	0

L_2

项集	支持度计数
$\{I_1, I_2\}$	4
$\{I_1, I_3\}$	4
$\{I_1, I_5\}$	2
$\{I_2, I_3\}$	4
$\{I_2, I_4\}$	2
$\{I_2, I_5\}$	2

选取大于最小支持度的项目集 →

由L_2产生C_3 →

C_3

项集
$\{I_1, I_2, I_3\}$
$\{I_1, I_2, I_5\}$

扫描D，对每一候选3项集计数 →

项集	支持度计数
$\{I_1, I_2, I_3\}$	2
$\{I_1, I_2, I_3\}$	2

选取大于最小支持度的项目集 →

项集	支持度计数
$\{I_1, I_2, I_3\}$	2
$\{I_1, I_2, I_3\}$	2

3.4 Apriori 算法源程序分析

```
#include<iostream>
#include<string>
#include<vector>
#include<map>
#include<algorithm>
using namespace std;

class Apriori
{
public:
    Apriori(size_t is =0,unsigned int mv=0)
    {
        item_size =is;
        min_value =mv;
    }
    //~Apriori() {};
    void getItem();
    map<vector<string>,unsigned int>find_freitem();  //求事务的频繁项
    //连接两个 k-1 级频繁项, 得到第 k 级频繁项
    map <vector<string>,unsigned int >apri_gen(unsigned int K, map<vector
        <string>,unsigned int>K_item);
    //展示频繁项集
    void showAprioriItem(unsigned int K,map< vector< string>,unsigned int >
       showmap);
private:
    map<int, vector<string>>item;                  //存储所有最开始的事务及其项
    map<vector<string>,unsigned int>K_item;        //存储频繁项集
    size_t item_size;                              //事务数目
    unsigned  int min_value;                       //最小阈值
};

void Apriori::getItem()                            //用户输入最初的事务集
{
    int ci=item_size;
    for (int i=0;i<ci;i++)
    {
        string str;
        vector<string>temp;
```

```
            cout<<"请输入第  "<<i+1<<"个事务的项集(123 end):";
            while (cin>>str && str !="123")
        {
            temp.push_back(str);
        }
        sort(temp.begin(),temp.end());
        pair<map< int, vector< string>>::iterator, bool> ret =item.insert(make_
          pair(i+1, temp));
        if (!ret.second)
        {
            --i;
            cout<<"你输入的元素已存在!请重新输入!"<<endl;
        }
    }
    cout<<"-------------运行结果如下:--------------"<<endl;
}

map<vector<string>,unsigned int>Apriori::find_freitem()
{
    unsigned int i=1;
    bool isEmpty=false;
    map<int, vector<string>>::iterator mit;
    for (mit=item.begin();mit!=item.end();mit++)
    {
        vector<string>vec=mit->second;
        if(vec.size() !=0)
            break;
    }
    if(mit==item.end())                          //事务集为空
    {
        isEmpty=true;
        cout<<"事务集为空!程序无法进行..."<<endl;
        map<vector<string>,unsigned int>empty;
        return empty;
    }
    while(1)
    {
        map<vector<string>,unsigned int>K_itemTemp=K_item;

        K_item =apri_gen(i++,K_item);

        if (K_itemTemp ==K_item)
        {
```

```
    i=UINT_MAX;
    break;
}
//判断是否需要进行下一次寻找
map<vector<string>,unsigned int>pre_K_item=K_item;
size_t Kitemsize=K_item.size();
//存储应该删除的第 K 级频繁项集，不能和其他 K 级频繁项集构成第 K+1 级项集的
//集合
if(Kitemsize!=1 && i!=1)
{
    vector<map<vector<string>,unsigned int>::iterator >eraseVecMit;
    map<vector<string>,unsigned int>::iterator pre_K_item_it1=pre_K_
      item.begin(), pre_K_item_it2;
    while (pre_K_item_it1!=pre_K_item.end())
    {
        map<vector<string>,unsigned int>::iterator mit =pre_K_item_it1;
        bool isExist=true;
        vector<string>vec1;
        vec1=pre_K_item_it1->first;
        vector<string>vec11(vec1.begin(),vec1.end()-1);
        while (mit!=pre_K_item.end())
        {
            vector<string>vec2;
            vec2=mit->first;
            vector<string>vec22(vec2.begin(),vec2.end()-1);
            if (vec11==vec22)
                break;
            ++mit;
        }
        if(mit==pre_K_item.end())
            isExist=false;
        if(!isExist && pre_K_item_it1!=pre_K_item.end())
            eraseVecMit.push_back(pre_K_item_it1);//该第 K 级频繁项应该
                                                  //删除
        ++pre_K_item_it1;
    }
    size_t eraseSetSize=eraseVecMit.size();
    if (eraseSetSize==Kitemsize)
        break;
    else
    {
        vector<map<vector<string>,unsigned int>::iterator>::iterator
```

```
                currentErs=eraseVecMit.begin();
            while (currentErs!=eraseVecMit.end())//删除所有应该删除的第 K 级
                                                  //频繁项
            {
                map<vector<string>,unsigned int>::iterator eraseMit=
                  *currentErs;
                K_item.erase(eraseMit);
                ++currentErs;
            }
        }
    }
    else
        if(Kitemsize ==1)
            break;
}
cout<<endl;
showAprioriItem(i,K_item);
return K_item;
}

map<vector<string>,unsigned int>Apriori::apri_gen(unsigned int K, map<vector
  <string>,unsigned int>K_item)
{
    if(1==K)                                    //求候选集 C1
    {
        size_t c1=item_size;
        map<int, vector<string>>::iterator mapit=item.begin();
        vector<string>vec;
        map<string,unsigned int>c1_itemtemp;
        while (mapit!=item.end())               //将原事务中所有的单项统计出来
        {

            vector<string>temp=mapit->second;
            vector<string>::iterator vecit=temp.begin();
            while (vecit!=temp.end())
            {
                pair<map<string,unsigned int>::iterator, bool>ret=c1_
                  itemtemp.insert(make_pair(*vecit++, 1));
                if (!ret.second)
                {
                    ++ret.first->second;
                }
```

```
        }
        ++mapit;
    }
    map<string,unsigned int>::iterator item_it=c1_itemtemp.begin();
    map<vector<string>,unsigned int>c1_item;
    while (item_it!=c1_itemtemp.end())       //构造第 1 级频繁项集
    {
        vector<string>temp;
        if(item_it->second >=min_value)
        {
            temp.push_back(item_it->first);
            c1_item.insert(make_pair(temp, item_it->second));
        }
        ++item_it;
    }
    return c1_item;
}
else
{
    cout<<endl;
    showAprioriItem(K-1,K_item);
    map<vector<string>,unsigned int>::iterator ck_item_it1=K_item.begin(),
      ck_item_it2;
    map<vector<string>,unsigned int>ck_item;
    while (ck_item_it1!=K_item.end())
    {
        ck_item_it2=ck_item_it1;
        ++ck_item_it2;
        map<vector<string>,unsigned int>::iterator mit=ck_item_it2;

        //取当前第 K 级频繁项与其后面的第 K 级频繁项集联合，但要注意联合条件
        //联合条件：两个频繁项的前 K-1 项完全相同，只是第 K 项不同，然后两个联合
        //生成第 K+1 级候选频繁项
        while(mit!=K_item.end())
        {
            vector<string>vec,vec1,vec2;
            vec1=ck_item_it1->first;
            vec2=mit->first;
            vector<string>::iterator vit1,vit2;

            vit1=vec1.begin();
            vit2=vec2.begin();
```

```
while (vit1<vec1.end() && vit2<vec2.end())
{
    string str1 = * vit1;
    string str2 = * vit2;
    ++vit1;
    ++vit2;
    if (K==2 || str1==str2)
    {
        if(vit1!=vec1.end() && vit2!=vec2.end())
        {
            vec.push_back(str1);
        }

    }
    else
        break;
}
if(vit1==vec1.end() && vit2==vec2.end())
{
    --vit1;
    --vit2;
    string str1 = * vit1;
    string str2 = * vit2;
    if (str1>str2)
    {
        vec.push_back(str2);
        vec.push_back(str1);
    }
    else
    {
        vec.push_back(str1);
        vec.push_back(str2);
    }
    map<int, vector<string>>::iterator base_item=item.begin();
    unsigned int Acount=0;
    while (base_item!=item.end() )//统计该 K+1 级候选项在原事务集
                                  //出现的次数
    {
        unsigned int count=0,mincount=UINT_MAX;
        vector<string>vv=base_item->second;
        vector<string>::iterator vecit, bvit;
        for (vecit=vec.begin();vecit <vec.end();vecit++)
        {
```

```
                        string t= * vecit;
                        count=0;
                        for (bvit=vv.begin();bvit <vv.end();bvit++)
                        {
                            if(t== * bvit)
                                count++;
                        }
                        mincount= (count <mincount? count: mincount);
                    }
                    if(mincount>=1 && mincount!=UINT_MAX)
                        Acount+=mincount;
                    ++base_item;
                }
                if(Acount >=min_value && Acount!=0)
                {
                    sort(vec.begin(),vec.end());
                    //该第 K+1 级候选项为频繁项，插入频繁项集
                    pair<map<vector<string>,unsigned int>::iterator,
                      bool>ret=ck_item.insert(make_pair(vec,Acount));
                    if (!ret.second)
                    {
                        ret.first->second +=Acount;
                    }
                }
            }
            ++mit;
        }
        ++ck_item_it1;
    }
    if (ck_item.empty())  //该第 K+1 级频繁项集为空，说明调用结束，把上一级频繁
                          //项集返回
        return K_item;
    else
        return ck_item;
  }
}
void Apriori::showAprioriItem(unsigned int K,map<vector<string>,unsigned int>
  showmap)
{
  map<vector<string>,unsigned int>::iterator showit=showmap.begin();
  if(K!=UINT_MAX)
      cout<<endl<<"第 "<<K<<" 级频繁项集为:"<<endl;
  else
```

```
        cout<<"最终的频繁项集为:"<<endl;
    cout<<"项  集"<<"  \t  "<<"频率"<<endl;
    while (showit!=showmap.end())
    {
        vector<string>vec=showit->first;
        vector<string>::iterator vecit=vec.begin();
        cout<<"{ ";
        while (vecit!=vec.end())
        {
            cout<<*vecit<<"  ";
            ++vecit;
        }
        cout<<"}"<<"  \t  ";
        cout<<showit->second<<endl;
        ++showit;
    }
}

unsigned int parseNumber(const char *str)//对用户输入的数字进行判断和转换
{
    if(str==NULL)
        return 0;
    else
    {
        unsigned int num=0;
        size_t len=strlen(str);
        for(size_t i=0;i<len;i++)
        {
            num *=10;
            if(str[i]>='0' && str[i]<='9')
                num +=str[i] -'0';
            else
                return 0;
        }
        return num;
    }
}

void main()
{
    //Apriori a;
    unsigned int itemsize=0;
    unsigned int min;
```

```
    do
    {
        cout<<"请输入事务数:";
        char * str=new char;
        cin>>str;
        itemsize=parseNumber(str);
        if (itemsize==0)
        {
            cout<<"请输入大于0正整数!"<<endl;
        }
    } while (itemsize==0);

    do
    {
        cout<<"请输入最小阈值:";
        char * str=new char;
        cin>>str;
        min=parseNumber(str);
        if (min==0)
        {
            cout<<"请输入大于0正整数!"<<endl;
        }
    } while(min==0);

    Apriori a(itemsize,min);
    a.getItem();
    map<vector<string>,unsigned int>AprioriMap=a.find_freitem();
    //a.showAprioriItem(UINT_MAX,AprioriMap);
    system("pause");
}
```

上述程序的运行结果如图3-1所示。输入事务数为9,最小阈值为2。9个事务的项集如下：$\{I_1,I_2,I_5\}$，$\{I_2,I_4\}$，$\{I_2,I_3\}$，$\{I_1,I_2,I_4\}$，$\{I_1,I_3\}$，$\{I_2,I_3\}$，$\{I_1,I_3\}$，$\{I_1,I_2,I_3,I_5\}$，$\{I_1,I_2,I_3\}$。程序首先搜索9个事务的项集,统计第1级备选项的支持度,如图3-1所示。将备选项的支持度与最小阈值作比较。由于最小阈值为2,因此不需要删除备选项,所得的第1级频繁项集为$\{I_1\}$,$\{I_2\}$,$\{I_3\}$,$\{I_4\}$,$\{I_5\}$。为产生第2级频繁项集,程序进行连接步,将第1级频繁项集与自身作连接得到第2级备选项,然后进行剪枝步,去除非频繁备选项,再比较所剩备选项的支持度与最小阈值,即得第2级频繁项集：$\{I_1,I_2\}$,$\{I_1,I_3\}$,$\{I_1,I_5\}$,$\{I_2,I_3\}$,$\{I_2,I_4\}$,$\{I_2,I_5\}$。按照上述步骤进行下去得到最终的频繁项集为$\{I_1,I_2,I_3\}$,$\{I_1,I_2,I_3\}$。

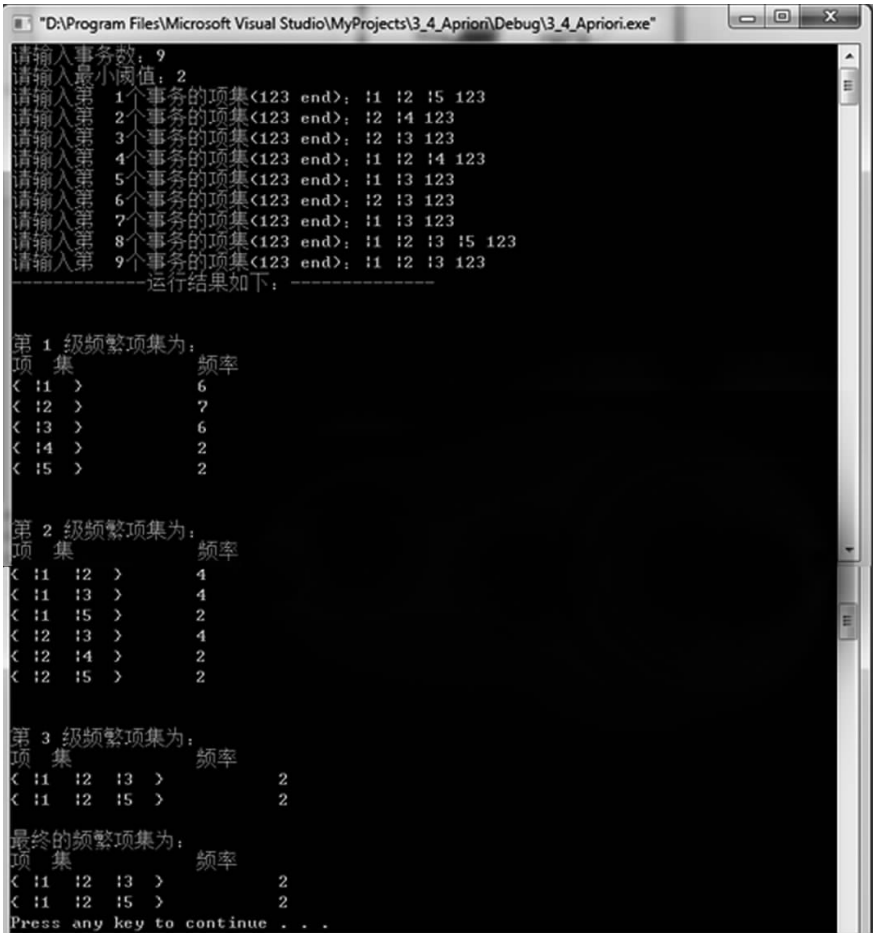


图 3-1 程序运行结果

3.5 Apriori 算法的特点及应用

3.5.1 Apriori 算法特点

Apriori 算法是应用最广泛的关联规则挖掘算法，它有如下优点。

① Apriori 算法是一个迭代算法。该算法首先挖掘生成 L_1，然后由 L_1 生成 C_2，再由 C_2 扫描事务数据库得到 L_2；根据 L_2 生成 C_3，由 C_3 扫描事务数据库得到 L_3，直到 C_k 为空而产生所有频繁项目集，Apriori 算法将生成所有大于等于最小支持度的频繁项目集。

② 数据采用水平组织方式。所谓水平组织就是数据按照{事务编号，项目集}的形式组织。

③ 采用 Apriori 优化方法。所谓 Apriori 优化就是利用 Apriori 性质进行的优化。

④ 适合事务数据库的关联规则挖掘。

⑤ 适合稀疏数据集。根据以往的研究，该算法只能适合稀疏数据集的关联规则挖掘，也就是频繁项目集长度稍小的数据集。

Apriori 算法作为经典的频繁项目集生成算法，在数据挖掘中具有里程碑的作用，但是随着研究的深入，它的缺点也暴露出来，主要有以下几个缺点。

① 多次扫描事务数据库，需要很大的 I/O 负载。在 Apriori 算法的扫描中，对第 k 次循环，候选集 C_k 中的每个元素都要扫描数据库一遍来验证其是否加入 L_k，假如一个频繁项目集包含 10 个项，那么就至少需要扫描数据库 10 遍。当数据库中存放大量的事务数据时，在有限的内存容量下系统 I/O 负载相当大，每次扫描数据库的时间就会很长，这样效率就会非常低。

② 可能产生庞大的候选集。Apriori 算法由 L_{k-1} 产生 k-候选集 C_k，其结果是指数增长的，例如 10^4 个 1-频繁项目集就有可能产生接近 10^7 个元素的 2-候选集。如此大的候选集对时间和内存容量都是一种挑战。

③ 在频繁项目集长度变大的情况下，运算时间显著增加。当频繁项目集长度变大时，支持该频繁项目集的事务会减少，从理论上讲，计算其支持度需要的时间不会明显增加，但 Apriori 算法仍然是在原来事务数据库中来计算长频繁项目集的支持度，由于每个频繁项目集的项目变多了，所以在确定每个频繁项目集是否被事务支持的开销也增大了，而且事务没有减少，因此频繁项目集长度增加了，运算时间也显著增加了。

3.5.2 Apriori 算法应用

Apriori 算法是应用最广泛的关联规则挖掘算法，通过对各种领域数据的关联性进行分析，挖掘成果在相关的决策制定过程中具有重要的参考价值。

Apriori 算法广泛应用于商业中，例如应用于消费市场价格分析中，它能够很快地求出各种产品之间的价格关系和它们之间的影响。通过数据挖掘，市场人员可以瞄准目标客户，采用个人股票行市、最新信息、特殊的市场推广活动或其他一些特殊的信息手段，从而极大地减少广告预算和增加收入。

Apriori 算法应用于网络安全领域，如入侵检测技术中。早期中大型的计算机系统中都收集了审计信息来建立跟踪档案，这些审计跟踪的目的多是为了性能测试或计费，因此对攻击检测提供的有用信息比较少。它通过模式学习和训练可以发现网络用户的异常行为模式，采用作用度的 Apriori 算法削弱了 Apriori 算法的挖掘结果规则，是网络入侵检测系统可以快速发现用户的行为模式，能够快速锁定攻击者，提高了基于关联规则的入侵检测系统的检测性。

Apriori 算法应用于高校管理中。随着高校贫困生人数的不断增加，学校管理部门资助工作难度也越来越大，数据挖掘算法可以帮助相关部门解决上述问题。例如，有的研究者将关联规则的 Apriori 算法应用到贫困助学体系中，并且针对经典 Apriori 挖掘算法存在的不足进行改进，先将事务数据库映射为一个布尔矩阵，用一种逐层递增的思想来动态地分配内存进行存储，再利用向量求“与”运算，寻找频繁项集。实验结果表明，这种改进后的 Apriori 算法在运行效率上有了很大的提升，挖掘出的规则也可以有效地辅助学校

管理部门有针对性地开展贫困助学工作。

Apriori算法被广泛应用于移动通信领域。移动增值业务逐渐成为移动通信市场上最有活力、最具潜力、最受瞩目的业务。随着产业的复苏，越来越多的增值业务表现出强劲的发展势头，呈现出应用多元化、营销品牌化、管理集中化、合作纵深化的特点。针对这种趋势，在关联规则数据挖掘中广泛应用的Apriori算法被很多公司应用。例如，依托某电信运营商正在建设的增值业务Web数据仓库平台，对来自移动增值业务方面的调查数据进行了相关的挖掘处理，从而获得了关于用户行为特征和需求的间接反映市场动态的有用信息，这些信息在指导运营商的业务运营和辅助业务提供商的决策制定等方面具有十分重要的参考价值。

3.6 小结

本章详细地介绍了关联规则挖掘的基本概念，对经典的关联规则挖掘算法—Apriori算法的原理以及发现频繁项目集的过程进行了描述，并用实例进行了说明，同时还分析了Apriori算法的特点和该算法存在的缺陷，得出它在发现频繁项集的过程中需要多次扫描事务数据库，此外还要产生大量的候选项集，这都会对算法的效率产生很大的影响，并且在频繁项目集长度变大的情况下，运算时间显著增加，最后介绍了Apriori算法在商业、网络安全、高校管理和移动通信等领域的应用。

思考题

1. 解释关联规则的定义。
2. 描述Apriori关联规则算法。
3. 如表3-3所示的数据库有5个事物。设min_sup=60%，min_conf=80%。

表3-3 数据库

TID	购买的商品	TID	购买的商品
I100	{M,O,N,K,E,Y}	I400	{M,U,C,K,Y}
I200	{D,O,N,K,E,Y}	I500	{C,O,O,K,I,E}
I300	{M,A,K,E}		

(1) 分别使用Apriori和FP增长算法找出所有频繁项集。比较两种挖掘过程的效率。

(2) 列举所有与下面的元规则匹配的强关联规则(给出支持度s和置信度c)，其中，X是代表客户的变量；item_i是表示项的变量(如A、B等)：

$$\forall x \in \text{transaction}, \text{buys}(X, \text{item}_1) \wedge \text{buys}(X, \text{item}_2) \Rightarrow \text{buys}(X, \text{item}_3)[s, c]$$

4. 如表3-4所示的关系表People是要挖掘的数据集，有三个属性(Age，Married，NumCars)。假如用户指定的min_sup=60%，min_conf=80%，试挖掘表3-4中的数量关

联规则。

表 3-4 关系表 People

RecordID	Age	Married	NumCars	RecordID	Age	Married	NumCars
100	23	No	0	400	34	Yes	2
200	25	Yes	1	500	38	Yes	2
300	29	No	1				

第 4 章　决策树分类算法

4.1　基本概念

4.1.1　决策树分类算法概述

从数据中生成分类器的一个特别有效的方法是生成一棵决策树(Decision Tree)。决策树表示方法是应用最广泛的逻辑方法之一,它从一组无次序、无规则的事例中推理出决策树表示形式的分类规则。决策树分类方法采用自顶向下的递归方式,在决策树的内部节点进行属性值的比较,根据不同的属性值判断从该节点向下的分支,在决策树的叶节点得到结论。所以,从决策树的根到叶节点的一条路径就对应着一条合取规则,整棵决策树就对应着一组析取表达式规则。

基于决策树的分类方法的一个最大的优点就是它在学习过程中不需要使用者了解很多背景知识,这同时也是它的最大缺点,只要训练例子能够用属性-结论式表示出来,就能使用该算法来学习。

决策树是一个类似于流程图的树结构,其中每个内部节点表示在一个属性上的测试,每个分支代表一个测试输出,而每个树叶节点代表类或类分布,树的最顶层节点是根节点。一棵典型的决策树如图 4-1 所示,它表示概念 buy_computer,预测顾客是否可能购买计算机。内部节点用矩形表示,而树叶节点用椭圆表示。为了对未知的样本分类,样本的属性值在决策树上测试。决策树从根到叶节点的一条路径就对应着一条合取规则,因此决策树容易转换成分类规则。

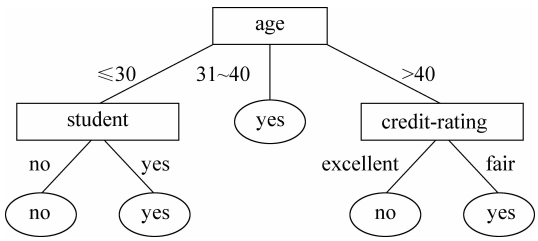


图 4-1　一棵典型的决策树

决策树是应用非常广泛的分类方法,目前有多种决策树方法,如 ID3、CN2、SLIQ、SPRINT 等,下面先介绍决策树分类的基本核心思想,然后详细介绍 ID3 和 C4.5 决策树方法。

4.1.2　决策树基本算法概述

决策树分类算法通常分为两个步骤:决策树生成和决策树修剪。